A Systematic Approach for Benchmarking of Container Orchestration Frameworks

Martin Straesser
University of Würzburg
Würzburg, Germany
martin.straesser@uni-wuerzburg.de

Jonas Mathiasch
University of Würzburg
Würzburg, Germany
jonas.mathiasch@informatik.uni-wuerzburg.de

André Bauer
University of Würzburg
Würzburg, Germany
andre.bauer@uni-wuerzburg.de

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

Container orchestration frameworks play a critical role in modern cloud computing paradigms such as cloud-native or serverless computing. They significantly impact the quality and cost of service deployment as they manage many performance-critical tasks such as container provisioning, scheduling, scaling, and networking. Consequently, a comprehensive performance assessment of container orchestration frameworks is essential. However, until now, there is no benchmarking approach that covers the many different tasks implemented in such platforms and supports evaluating different technology stacks. In this paper, we present a systematic approach that enables benchmarking of container orchestrators. Based on a definition of container orchestration, we define the core requirements and benchmarking scope for such platforms. Each requirement is then linked to metrics and measurement methods, and a benchmark architecture is proposed. With COFFEE, we introduce a benchmarking tool supporting the definition of complex test campaigns for container orchestration frameworks. We demonstrate the potential of our approach with case studies of the frameworks Kubernetes and Nomad in a self-hosted environment and on the Google Cloud Platform. The presented case studies focus on container startup times, crash recovery, rolling updates, and more.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computer systems organization** → **Cloud computing**; • **Networks** → Network performance analysis.

KEYWORDS

container orchestration; benchmarking; kubernetes; nomad; performance

1 INTRODUCTION

Container orchestration (CO) frameworks are the engines of modern cloud environments. According to the CNCF Annual Survey 2021 [9], 96% of surveyed organizations and 5.6 million developers are using or evaluating Kubernetes, the most popular container orchestration framework. CO frameworks act as an abstraction layer for access to a computing cluster and can have different roles. In cloud-native computing, developers interact directly with CO frameworks and use them for container deployment, maintenance, storage, and more. In serverless computing, they are the backbone of serverless platforms, responsible for critical tasks such as container provisioning, placement, and scaling. In either use case, CO frameworks fulfill multiple performance-critical tasks. From a performance engineering perspective, it is therefore essential to evaluate frameworks like Kubernetes because they significantly impact the performance of cloud applications, as shown in several studies [24, 26, 33, 34].

Both conceptual and technical challenges arise for comprehensive performance evaluation and benchmarking of container orchestration frameworks. Conceptually, a clear benchmarking scope has to be established, which is not trivial as the term container orchestration is not used uniformly. However, a wide range of tasks is usually associated with CO frameworks, for example, load balancing, networking, scaling, scheduling, and availability. All these tasks influence each other and come with different metrics for which measurement methods have to be defined. Another challenge is the interference between the application and orchestrator performance. On the one hand, we want to look at the performance of CO frameworks as isolated as possible. On the other hand, we do not want to lose sight of the application, which, in practice, strongly influences end-to-end metrics like response times that are crucial for users. From a technical point of view, one problem is that different frameworks have very different technology stacks and interfaces. Existing benchmarking approaches are limited to analyzing single orchestration tasks or only considering one specific CO framework and technology stack. To the best of our knowledge, there is currently

no approach that supports benchmarking multiple orchestration frameworks and covers the plurality of container orchestration tasks.

In this paper, we take a systematic approach to benchmarking of container orchestration frameworks. Based on a review of different definitions of container orchestration, we derive a meaningful benchmarking scope and use cases. Next, we define generic core requirements for CO frameworks and associate them with specific performance metrics. We then design an orchestrator-agnostic benchmark architecture for evaluating these metrics. Our benchmarking framework COFFEE implements the designed architecture and allows the definition of complex benchmarking campaigns through a user-friendly script-like language. We use COFFEE to analyze and compare the performance of the CO frameworks Kubernetes and Nomad in a self-hosted environment as well as on the Google Cloud Platform.

The goal of this paper is to bring the topic of benchmarking container orchestration frameworks into the public eye and enable comprehensive performance evaluation through orchestrator-agnostic benchmarking methodology and framework. The proposed metrics and the framework developed in this work can help users to decide which orchestration framework to use depending on their use case and compare different configuration options. In summary, the contributions of this paper are:

- We define a benchmarking scope for container orchestration frameworks, including core requirements, performance metrics, workloads, and methodology;
- We present a benchmark architecture and COFFEE, a flexible and extensible tool for benchmarking of CO frameworks, as an implementation of our approach;
- We provide empirical evidence for the usability of our approach by conducting case studies in two different environments with Kubernetes and Nomad as frameworks under test.

The remainder of this paper is structured as follows: In Section 2, we give some background on the term container orchestration and the two frameworks evaluated in this work. In Section 3, we derive requirements for container orchestration frameworks and associate metrics with every requirement. Furthermore, we present a benchmark architecture for container orchestration frameworks. Section 4 presents our benchmarking framework COFFEE, which implements the proposed architecture. Section 5 then features several case studies, including Kubernetes and Nomad as frameworks under test. We discuss the limitations of our work and open challenges in Section 6. Section 7 summarizes related work and outlines shortcomings. Finally, Section 8 concludes the paper.

2 BACKGROUND

This section focuses on the term container orchestration and introduces Kubernetes and Nomad, two state-of-the-art CO frameworks investigated in this work. There is not one commonly accepted and highly cited definition of the term container orchestration. Casalicchio [6] states that container orchestration is concerned with managing containers at runtime and supporting deployment, execution, and maintenance. Common features are resource limit control, scheduling, load balancing, health check, fault tolerance,
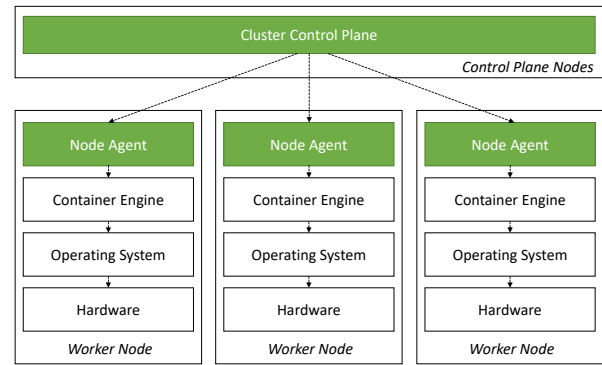

Figure 1: Components of a container cluster with container orchestration (green).

and autoscaling. Khan [18] names seven capabilities of container orchestration platforms: cluster state management and scheduling, high availability and fault tolerance, security, networking, service discovery, continuous deployment, monitoring, and governance. Rodriguez and Buyya [28] propose a container orchestration reference architecture with the key tasks of container provisioning, monitoring, scheduling, as well as accounting and admission control. As over the past years, CO frameworks experienced massive growth in usage and community development, none of these definitions covers the full feature bandwidth of modern platforms.

Looking at non-scientific sources and analyzing the definitions from technology leaders like IBM [14], VMware [10], and others, we found the definition of Red Hat [15] especially helpful, as it matches the features of modern frameworks well. It states that container orchestration automates the deployment, management, scaling, and networking of containers. The following typical tasks are named: provisioning and deployment, configuration and scheduling, resource allocation, container availability, scaling, load balancing and traffic routing, health monitoring, application configuration, and securing container interaction. This definition overlaps with the ones from the scientific literature but has two advantages in particular. It introduces container orchestration as a generic and broad term but also names concrete responsibilities of modern CO frameworks. We use this definition as a basis for deriving the scope of benchmarking we target in this work.

From a technical point of view, CO frameworks represent an abstraction layer for accessing a cluster capable of running containerized applications. Figure 1 schematically shows the main components of a container cluster. Usually, a control plane (e.g., a set of master nodes) interacts with several other nodes (worker nodes) via a node agent. The worker nodes run a container engine (e.g., containerd[1]). The container engine uses kernel functions or low-level software like *cgroups* to allocate resources.

We use two container orchestration frameworks in our case studies. Our first framework under test is Kubernetes,[2] which is the current market leader [9]. Initially developed by Google, Kubernetes nowadays has a large community. Containers in Kubernetes

---

[1]https://containerd.io/
[2]https://kubernetes.io/

are organized in so-called pods, and each pod is assigned an IP for networking. Kubernetes uses a declarative interaction concept where desired states (like the number of running service instances) are constantly compared to observed states. Our second framework under test is Nomad,[3] a container orchestration platform developed by HashiCorp. Nomad supports both containerized and non-containerized applications. It uses Consul[4] to enable service networking. The equivalents to pods in Kubernetes are so-called tasks. Compared to Kubernetes, Nomad claims more simplicity and high scalability, supporting clusters of over 10,000 managed nodes. The Nomad website [25] lists companies that use the framework in production.

3 BENCHMARKING APPROACH

In this section, we explain our approach to enable benchmarking of CO frameworks. First, we define the use cases and the benchmarking scope. We then formulate core requirements for CO frameworks and link performance metrics to each requirement. The section ends with our proposed benchmark architecture, which allows one to measure and evaluate the performance of CO frameworks.

3.1 Benchmarking Use Cases and Scope

Various implications and use cases for benchmarking are created from the technical scheme of container orchestration frameworks presented in Figure 1. When we evaluate the performance of a CO framework, we automatically examine the entire technology stack; that is, the software and hardware of the nodes also play a role. This has to be considered when interpreting benchmarking results. In general, we identify three use cases for the benchmarking of container clusters:

1. Comparing different CO frameworks: In this case, a fixed resource landscape is given, and different CO frameworks are evaluated in this context. The use case in practice is to choose a CO framework for a specific environment.

2. Comparing configuration options of one CO framework: Similar to the first use case, the environment is not changed. Instead, a different configuration of the CO framework is used (e.g., a different scheduling algorithm or networking solution). The use case in practice is fine-tuning a specific framework (e.g., comparing different networking plugins for Kubernetes).

3. Comparing different cluster environments: In this case, the focus is not on the CO framework but on the nodes. There can be different configurations of the nodes (e.g., container engines, operating systems, or hardware resources). One use case in practice is selecting the size of VM instances in public clouds.

We focus on use cases 1 and 3 and present respective case studies in Section 5. To enable benchmarking of CO frameworks, we have to define a benchmarking scope, that is, the (performance-relevant) tasks a CO framework is assumed to fulfill that are subject to evaluation through benchmarking. In Section 2, we discussed that the term container orchestration is not clearly defined. However, we also discovered overlaps between different definitions and associated tasks of CO frameworks. In this paper, we assume the following performance-critical tasks of CO frameworks:

- T1: Container provisioning and deployment
- T2: Container scheduling
- T3: Resource allocation
- T4: Container availability
- T5: Health monitoring
- T6: Scaling
- T7: Load balancing and traffic routing
- T8: Inter-container networking

This list correlates strongly with the definition by Red Hat [15]. However, we omit the security properties of CO frameworks, as they are hard to generalize across different frameworks. Given this list of assumed tasks of CO frameworks and the environment setting from Figure 1, three necessary properties for our benchmarking approach emerge. First, it must provide a level playing field for evaluating different CO frameworks. This is implied from use case 1 and goes along with the best practice of designing a fair benchmark that is not tailored to a specific system under test [20]. Second, since we are dealing with different nodes and technology stacks, the measured performance metrics must be generic, with minimal assumptions on the orchestrator and cluster nodes. Third, the variety of tasks of a CO framework necessitates broad coverage and metrics for every task. One should also consider that all of these tasks are concurrently executed, and therefore interdependencies between individual tasks exist. A benchmarking approach should therefore target as many tasks as possible in parallel instead of evaluating them one after the other.

3.2 Core Requirements and Metrics

Now that we have defined the scope of our benchmarking effort and the tasks to be tested, we need to decompose the abstract list of tasks into concrete fine-granular requirements, which can then be evaluated and quantified by specific metrics. Table 1 shows an overview of seven core requirements for CO frameworks and which tasks are addressed by these requirements. In the following, we detail these requirements and give metrics that can quantify how well a CO framework fulfills each requirement. The metrics should be easily measurable and not impose any assumptions on the nodes or the orchestration framework.

R1. One of the core requirements of any CO framework is the ability to start containers. If a new container has to be launched, the scheduling algorithm decides first on which node the container should be placed. Then, node resources are reserved, and if necessary, the image is downloaded from a registry before the container is started. The main metric to quantify the start performance is the readiness time, that is, the time it takes from issuing the start command until the container context is initialized and the container is running. The readiness time consists of different phases: scheduling, image pull, and start time. The start time is the time it takes for the container engine to process the image manifest and set up the controller environment (e.g., control groups and namespaces).

R2. Every container that was started should also be terminated and removed. The associated performance metric is the removal time. It is to be expected that the removal time is significantly lower than the readiness time since no placement and image pull take place. However, depending on the container, there may also

Martin Straesser, Jonas Mathiasch, André Bauer, & Samuel Kounev

Requirement	T1	T2	T3	T4	T5	T6	T7	T8
R1: Containers can be started.	X	X	X		X			
R2: Containers can be removed.	X							
R3: Containers can be restarted manually or in case of failures.	X	X	X	X	X		X	
R4: Containers can be updated to a new version.	X	X	X	X	X			
R5: Provisioned resources can be varied depending on workload.	X	X	X		X	X		
R6: Requests from external sources are balanced across running containers.							X	
R7: Containers are able to communicate with other containers in the same cluster.								X

Table 1: Core requirements for container orchestration frameworks and related tasks from Section 3.1.

be pre-destroy statements that allow a graceful exit (e.g., a clean disconnect from a database).

R3. Automated restarts are usually out of scope for container engines but a core task of CO frameworks. The reasons for restarts can be various: developers can manually trigger restarts, or unexpected errors in the container process might cause them. Another reason could be that the container runs out of resources (e.g., if it is overloaded). In the latter case, health monitoring should track the state of the container. One performance metric for this requirement is the restart time. It comprises the removal time of the old container and the readiness time of the new one. In case of an error-induced restart, the failure discovery time can be considered, that is, the time difference between the occurrence of an error and the time when the restart is initiated. If the container receives external requests, the load balancing algorithm must also react and not allocate any additional load to the container. In this case, the number of failed requests can also be considered a performance metric.

R4. In continuous deployment, it is essential to be able to perform a rolling update, that is, to update a set of containers to a new version. One common use case is to change the container image. The task of the CO framework is to perform and coordinate this update as fast as possible but also to maintain availability and avoid violating service-level agreements. Consequently, we use the total update time alongside the response time and the number of failed user requests as performance metrics.

R5. As mentioned in Section 3.1, scaling is one of the main tasks of CO frameworks. Dynamic resources can be the number of container instances (horizontal scaling), the computing resources allocated to a container (vertical scaling), or the number of nodes in the cluster (cluster scaling). Typically, autoscalers are evaluated using a mixture of cost and QoS metrics such as response times [13, 31].

R6. In the context of microservice applications, usually multiple application instances are deployed. When a user requests a service, the request must be assigned to a chosen instance. Load balancing is used to avoid overloading and performance degradation. Metrics for the evaluation of load balancers are the end-to-end response times of user requests and the load balancer's generated overhead [39]. In this paper, we also consider the distribution of requests over service instances to determine how the load balancer distributes requests over a set of instances.

R7. Communication between containers is essential for multiservice applications. However, also for containers of the same kind, container-to-container communication can be essential (e.g., when states have to be synchronized). Usually, this kind of networking is realized using overlay networks. Classical network metrics like
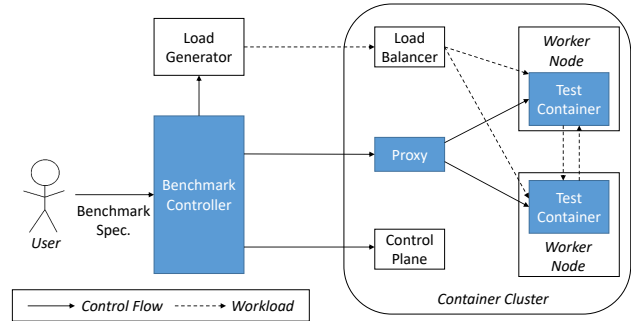

Figure 2: Benchmark architecture and components (blue).

throughput, round-trip time, and latency can be used to evaluate in-cluster networking.

3.3 Benchmark Architecture

In this section, we propose an architecture that can quantify a CO framework's performance based on the requirements and associated metrics defined above. As mentioned in Section 3.1, the architecture should be as generic as possible and not depend on the specifics of individual orchestration frameworks and underlying technology stacks.

Figure 2 shows our proposed architecture. The core component is the benchmark controller, which receives the benchmark specification from the user. Furthermore, test containers are deployed in the cluster. Individual instructions within the benchmark specification can either generate requests to the cluster control plane (e.g., for container starts) or selected test containers (e.g., for failure injection). In the latter case, a proxy is needed. This is because individual containers are usually not directly addressable from outside as a network barrier between cluster and external users exists. A load generator is used to imitate user requests. The benchmark controller and load generator should be deployed outside the cluster to avoid interferences. Several test container instances communicate with each other via in-cluster networking.

This architecture satisfies all the properties we defined earlier. The only assumptions made about the CO framework are that a proxy can be deployed, the controller has access to the control plane, and a load generator can send requests to the cluster. Modern CO frameworks fulfill all these requirements. Furthermore, the proposed architecture allows us to measure many of the performance metrics from Section 3.2. The controller generates a timestamp

Command	Description
START <n>	Starts n test container instances
RESTART <n>	Restarts n instances
HEALTH <n>	Sets unhealthy flag in n instances
CRASH <n>	Causes crash of n instances
UPDATE <n>	Updates n instances (changes image)
NETWORK	Measures round-trip time between running instances
REMOVE <n>	Shuts down n instances
LOAD / ENDLOAD	Starts/Ends load generation
SEQ / ENDSEQ	Starts/Ends a sequence
LOOP <n> / ENDLOOP	Starts/Ends a loop with n iterations
OFFSET <t>	Invokes next command after t sec.
DELAY <t>	Pauses a sequence/loop for t sec.

Table 2: Commands for test campaign definition.

when the command is sent for the measurements of readiness, removal, update, and outage times. The created, removed, or updated test container instance also reports a timestamp, and the controller can calculate the time difference between the two events. Using a load generator enables measuring metrics like response times, the number of failed user requests, and more. Further fine-granular metrics might be requested via the cluster control plane. The built-in user-level metrics might be complemented with system-level metrics, like CPU utilization of the nodes. However, note that no extended monitoring capabilities are necessary to use the proposed benchmark architecture.

4 BENCHMARKING FRAMEWORK

This section introduces COFFEE, a benchmarking framework for container orchestrators. Our benchmarking concept and reference architecture served as a basis for the design of COFFEE. Accordingly, it consists of 3 main components: controller, test container, and proxy. All components use Java and Spring as implementation technologies. The source code of COFFEE and all examples used in this paper can be found on GitHub[5] and Zenodo.[6]

The controller is the most complex component of COFFEE, as Figure 3 shows. The controller requires two inputs. First, the test campaign must be specified. The textual input is translated into a set of operations. All currently supported operations can be found in Table 2. In general, there are three types of operations. The first group interacts with the load generator (LOAD, ENDLOAD). COFFEE provides an interface for the HTTP Load Generator,[7] but other load generators can be integrated with low effort. The second group consists of orchestrator-agnostic operations forwarded to the test containers via the proxy (CRASH, HEALTH, NETWORK). These are processed by the test containers and therefore do not need to be re-implemented for different orchestrators. The third group of operations is orchestrator-specific and needs to be implemented for each framework. It includes container starts (START), removals (REMOVE), updates (UPDATE), and manual restarts (RESTART).
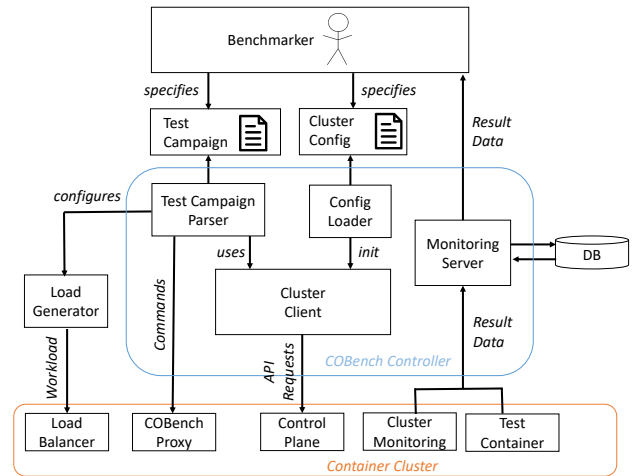
---

[5] https://github.com/DescartesResearch/COFFEE
[6] https://doi.org/10.5281/zenodo.7603961
[7] https://github.com/joakimkistowski/HTTP-Load-Generator

Figure 3: COFFEE controller in detail.

Furthermore, auxiliary commands enable the definition of complex test campaigns. By default, all commands are executed asynchronously and in parallel; by specifying an OFFSET, the user can specify when a command should be sent. If a sequential execution of a series of commands is desired, one can wrap this sequence by SEQ/ENDSEQ or LOOP/ENDLOOP for repeated executions. A command is considered completed if all expected responses/metrics of the command have been reported to the controller. For example, if five containers need to be started, the controller waits to continue a sequence until five start times have been reported. The keyword DELAY can be used within a sequence to set a pause between operations.

```
LOOP 100
    START 10
    DELAY 20
    LOAD
        DELAY 120
        CRASH 10
        DELAY 120
    ENDLOAD
    REMOVE 10
    DELAY 30
ENDLOOP
```

Listing 1: Example test campaign for failure recovery.

Listing 1 shows an example test campaign, which is also used in Section 5.3. Here, we use LOOP 100 to indicate that this experiment should be repeated 100 times. First, we start ten test containers. Afterward, we wait 20 seconds until the load generator is started. The configuration of the load generator contains many parameters, e.g., request rates over time and target IPs, and is done not within the test campaign but dynamically at runtime. After starting the load generation, the test script is paused for 2 minutes. This is the warmup phase for the test containers, as the load generator sends requests in parallel. In the next step, a CRASH command is issued, which terminates all ten running containers. We again wait 2 minutes while the load generator continues to record the

response times during that recovery phase. The load generator is shut down, and all ten containers are removed at the end of one experiment run. The 30 seconds delay at the end provides a short pause before the next loop iteration.

The specified test script can be reused for different frameworks. However, the controller also needs an orchestrator-specific configuration containing at least the cluster control plane address, the orchestrator type, and authentication data if necessary. Currently, COFFEE supports Kubernetes and Nomad as frameworks under test. COFFEE can be extended to work with other orchestrators as well. Therefore, the orchestrator-specific configuration, the four orchestrator-specific commands explained above, and some query operations, like the retrieval of node names, must be implemented. The controller is also concerned with processing the experiment results. It receives result data from the cluster and writes them to a result database. A summary of the most important results can be printed on the console at the end of the experiment.

The second core part of COFFEE consists of the test containers deployed in the cluster. Once the test container's context is initialized, a timestamp is taken and sent to the controller so that it can confirm the instance start and determine the readiness time. Once the test container is started, six endpoints can be accessed. The *load* endpoint is accessed by the load generator and performs some dummy operations that can be freely implemented. In our case, it creates CPU load by checking prime numbers and doing factorial calculations with Java's BigInteger class. The *load* endpoint also contains a counter tracking the number of received requests from the load generator. The second endpoint *crash* causes the process to end with a non-zero exit code. The third endpoint is used by the container orchestrator for HTTP-based health monitoring. The fourth endpoint is called in case of a HEALTH command and causes the HTTP-based health monitoring to fail. The last two endpoints are used for NETWORK tests. One is called when the instance should send requests to other containers, while the other is used for receiving requests from other instances. Before the container is terminated, it sends a timestamp to the controller to confirm the instance removal. In this step, the number of received load requests is also submitted to the controller.

The proxy application acts as a reverse proxy and forwards the commands HEALTH, NETWORK, and CRASH to specific containers. It receives the target addresses from the controller. Overall, COFFEE offers a simple design and is easy to use. It enables automatized benchmarking experiments with different CO frameworks. By design, COFFEE takes benchmark scalability into account by reducing the number of requests to the controller to a minimum. As COFFEE makes no critical assumptions on the test cluster, it can be used with test clusters of different sizes and resources. The proxy and test containers can be scaled both horizontally and vertically if necessary. The scalability of the load generation depends on the used generator. The HTTP Load Generator used in this work supports distributed execution on various nodes to achieve high request rates.

5 CASE STUDIES

In this section, we use COFFEE for some selected case studies. We compare the CO frameworks Nomad and Kubernetes in a self-hosted setup and on virtual machines running in the Google Cloud. The test scenarios include container provisioning and networking, failure recovery, and rolling updates. Before that, we present the technical setup in more detail. All measurement results and evaluation scripts are available in our replication package.[8]

5.1 Technical Setup

Each test cluster consists of three worker nodes and one node acting as the control plane. In the case of our self-hosted environment, the CO frameworks run on bare-metal servers of type HPE Pro-Liant DL360 Gen9 with Intel(R) Xeon(R) CPU E5-2650 v4 and 16 GiB DDR4 RAM and Ubuntu 18.04.6 LTS as the OS. In the Google Cloud, we use VM instances of type e2-standard-8 with Ubuntu 22.04 LTS placed in one common compute zone. The COFFEE controller, result databases, and load generators run on independent machines outside the test cluster to avoid interferences. We use Kubernetes version v1.24.4 in the self-hosted setup and the equivalent v1.24.4-gke.800 in the Google Kubernetes Engine (GKE), both with containerd v1.6.6 as the container engine. For container networking, we use Flannel v0.19.2 as the networking plugin in our self-hosted environment, while GKE provides a built-in networking solution. For Nomad, we use version v1.4.1 with Docker v20.10.18 and Consul v1.13.2. Nomad also requires the manual deployment of a load balancing solution. For this, we use HAProxy v2.6.2. Note that to the best of our knowledge, Google Cloud does not provide a managed Nomad service, and it is, therefore, necessary to deploy it manually on a set of Google Cloud VMs. We provide the scripts used to setup our test nodes in our GitHub repository[5].

5.2 Container Provisioning and Networking

This section focuses on three core requirements for CO frameworks: container starts, container removals, and inter-container networking. We examine readiness and removal times in Kubernetes and Nomad running in our self-hosted environment and the Google Cloud. As an additional degree of freedom, we consider the number of containers n that should be started simultaneously. In both Kubernetes and Nomad, declarative methods are used for deployments (Kubernetes) and tasks (Nomad) to specify the desired number of running replicas. A warmup run ensures that the test image is already present on all cluster nodes;therefore the pull time does not matter in the experiments. The experiment is rather simple: n containers are deployed, and after 30 seconds, n containers are removed again. Between two repetitions, there is also a delay of 30 seconds. We repeat the experiment 100 times in all test environments.

Figure 4a shows the average readiness time per container for the four test systems. The number of containers to be started is plotted on the horizontal axis. The error bars indicate the 95 percent confidence interval (CI) in this and all the following figures. As we perform 100 repetitions, we can assume that the sample variance is a good estimation of the population variance, and therefore we can use the sample variance to calculate a CI [20]. Figure 4a shows that regardless of the CO framework, the self-hosted environment

---

[8]https://doi.org/10.24433/CO.8875394.v3
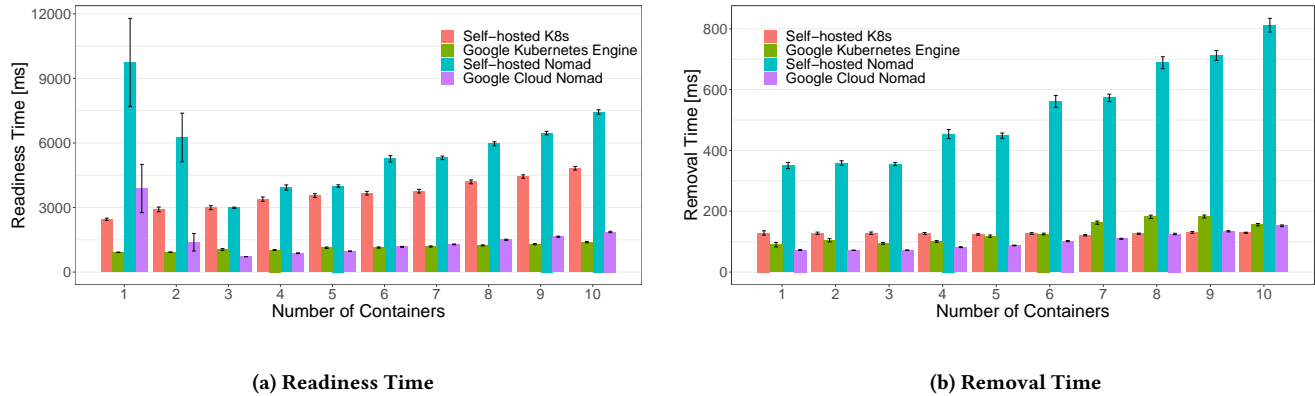
(a) Readiness Time

(b) Removal Time

Figure 4: Container provisioning metrics for different CO frameworks and test clusters.

with the older technology stack performs worse than the Google Cloud VMs. This can be seen in the pairwise comparison between the red and green, as well as blue and purple bars. Furthermore, the average readiness time per container increases with the number of desired containers. This is probably due to queueing effects during the scheduling process. The only anomalies are the two Nomad test systems, which show high readiness times with high variability for low replica numbers. Since this effect repeatedly occurs in both test environments, it is specific to the CO framework, not the environment. The investigation of the readiness times shows that COFFEE can investigate differences between CO frameworks (use case 1 from Section 3.1) as well as differences between two test clusters (use case 3).

In contrast, no clear trends can be seen in the average removal times per container shown in Figure 4b. Looking at all four test systems, there is no consistent correlation between the number of containers to remove and the removal time. However, there are still two takeaways. First, the removal time is significantly lower than the readiness times, which is in line with our expectations. Furthermore, we see that self-hosted Nomad performs worst among the test systems.

As a second baseline test for our clusters, we look at the performance of the in-cluster networking. We deploy one test container per worker node (3 in total). After a 30-second pause, requests are exchanged between all containers. Thereby, each instance acts as both sender and receiver. This creates six evaluated routes in total. As a performance metric, we consider the round-trip time (RTT) of the messages. After the end of the network tests, the three test instances are shut down again. The experiment is repeated 100 times, with a 30-second pause between the runs. As an additional test system, we created a multi-region Kubernetes cluster in the Google Cloud. One node is located in the compute region us-west1 (Oregon, US), one in europe-west2 (London), and one in asia-southeast1 (Singapore). The node specification and software stack are the same as described for the other Google Cloud VMs. We expect to measure a significantly higher round-trip time than in the self-hosted environment and the single-region GKE and Google Cloud Nomad clusters.

Test system	Avg. round-trip time [ms]
Self-hosted K8s	15.07 ± 0.60
Google Kubernetes Engine	24.63 ± 3.46
Self-hosted Nomad	15.17 ± 0.52
Google Cloud Nomad	14.95 ± 0.46
Multi-Region Cluster	396.52 ± 11.47

Table 3: Average round-trip times between two nodes.

Table 3 shows all test systems' average round-trip times and confidence intervals. Since the RTT does not vary significantly between different routes for our four standard test systems, all routes are included in the calculation of the average, i.e., a total of 600 measured values. It can be seen that the RTT in the self-hosted and the Google Cloud Nomad clusters are nearly equal. In the case of the GKE cluster, we observe slightly higher values. However, it should be noted that the networking metrics are subject to measurement errors caused by imperfect time synchronization between the hosts, as discussed in Section 6.

Furthermore, our hypothesis that the round-trip times in the multi-regional cluster are significantly higher than in the other systems is confirmed. Table 3 reports the average value over all routes, but we also measure significant differences between the routes in this case. For example, the Oregon-London route was the fastest, with an average RTT of 267.88ms, followed by the Oregon-Singapore route with 326.62ms. The slowest route was London-Singapore, with a 595.07ms average round-trip time. In summary, we demonstrated that our approach is able to detect differences in network performance. Further interesting investigations could be the differences between different container networking solutions, similar to the works of Bankston and Guo [2] and Zeng et al. [40]. This would match our benchmarking use case 2 from Section 3.1.

5.3 Failure Recovery

After the baseline tests from the previous section, we shift our focus to more complex experiments. In this section, we examine how fast our test systems can detect and react to the failure of several containers. We have already introduced the test procedure as an
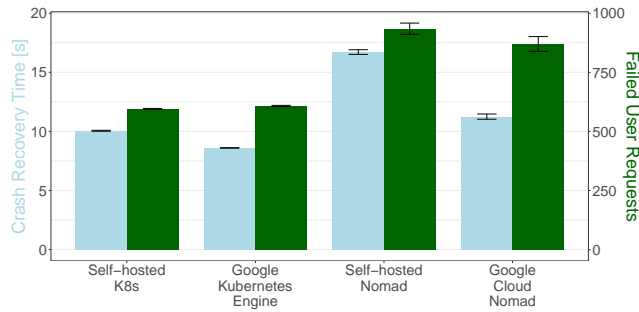
Figure 5: Failure recovery metrics.

Figure 6: Rolling update metrics.

example in Listing 1. In summary, ten containers are started and stressed with 50 user requests per second using a load generator. After a warmup period of two minutes, all containers are abruptly terminated, i.e., the container process exits with a non-zero code. In parallel, requests are sent at the same rate. We repeated the experiment 100 times. In all test environments, HTTP-based container health checks are configured in intervals of 10 seconds. The expectation is that after detecting the container crashes, all ten containers will be restarted. Requests should fail for a short time as all containers crash almost simultaneously.

We consider the crash recovery time, i.e., the time from the crash of the application to the point where all ten instances are restarted and ready, as a performance metric. Furthermore, we look at the total number of failed user requests reported by the load generator. Figure 5 shows our measurement results. We see for all four test systems that the number of failed requests correlates strongly with the crash recovery time. This was to be expected due to the constant request rate. Furthermore, we see that Kubernetes has lower crash recovery times than the Nomad systems in pairwise comparison within the same environment and in absolute comparison.

The order of magnitude of the measured values is in line with our expectations considering a simplified model. Let us assume that the time of the crash and the time of the next HTTP-based health check are independent. With a deterministic health check interval of ten seconds, the expected time until the next health check is five seconds. Suppose we add the removal and readiness time from Section 5.2 for ten parallel starting containers and a little overhead to this value. In that case, the result corresponds approximately to the observed values here. Assuming that the health checks work reliably, the readiness time of the containers is again one of the most critical performance factors. Our methodology, as seen from the confidence intervals, also ensures good repeatability of the experiments containing container crashes.

5.4 Rolling Updates

In this section, we examine performance metrics related to rolling updates. The experiment setup is similar to the previous experiment. Ten containers are started and stressed with a request load of 50 requests per second (req/s). After a 2-minute warmup phase, all containers are updated to a new version; that is, the image is changed. To prevent downtimes by terminating all containers at the
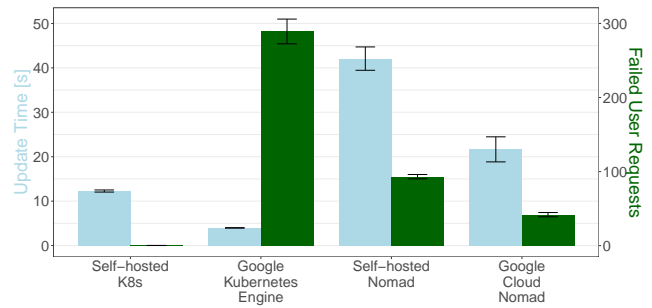
same time, both Nomad and Kubernetes offer settings for rolling updates. Kubernetes provides two key parameters. First, *maxUnavailable* specifies the percentage of running replicas that can be down at the same time during the update process. Second, *maxSurge* specifies how many additional containers can be started during an update relative to the number of desired replicas. We set *maxSurge* and *maxUnavailable* to 25%, meaning between 8 and 12 pods are deployed during an update. In Nomad, one can set the number of containers that can be updated simultaneously (absolute number). In our experiments, we set this parameter named *maxParallel* to 2, similar to the *maxUnavailable* setting in Kubernetes. In our experiments, a warmup run ensures that both the original image and the update image are already present on all nodes; that is, no image pull takes place.

We consider two performance metrics for rolling updates. First, the update time is the time it takes until all ten test containers have been updated and become ready. Second, we look at the number of failed user requests, similar to Section 5.3. Figure 6 shows our measurement results for the four test systems. Similar to the previous experiments, the Google Cloud VMs perform significantly better considering pairwise comparison with the self-hosted environment. We see an excellent update performance in the self-hosted Kubernetes environment with an update time of 10 to 15 seconds and almost no failed requests (average below 1). Google Kubernetes Engine offers the fastest update with an update time well below 10 seconds, but up to 300 user requests fail, corresponding to a downtime of about 6 seconds. The total update times for Nomad are significantly higher than for Kubernetes. Furthermore, a small but measurable number of requests fail for those test systems.

The results obtained require a more detailed analysis. Figure 7 shows two exemplary runs for Google Kubernetes Engine and Google Cloud Nomad. The number of failed requests over time is shown in red. In addition, we depict the number of ready instances, that is, instances registered in the COFFEE controller, in black. If the number of ready instances decreases, a container of the old version is shut down. If the number of ready instances increases, a container with the new image is deployed. In Figure 7a, we see that the update for the Google Kubernetes Engine is very fast. The time between the first event (the removal of the first container) and the last event (the removal of the last container) is just about four seconds. Furthermore, we see that requests fail mainly towards or

Test system	Avg. load rate [req/s]	Std. Dev.
Self-hosted K8s	4.567	1.777
Google Kubernetes Engine	4.565	1.775
Self-hosted Nomad	4.624	0.352
Google Cloud Nomad	4.431	0.618

Table 4: Received user requests per second and instance.

after the end of the update process, that is, when all old containers have been terminated. This can be explained by the fact that some requests are still routed to IP addresses that are no longer occupied. We further see that there are always at least 8, but at some points, more than the expected 12 containers deployed and ready. There are two possible causes for this unexpected behavior: First, the Kubernetes documentation states that terminating pods do not belong to the available quantity and, therefore, more pods than expected can be deployed during the update.[9] Second, it can also be a matter of measurement errors, as the event are separated by only a few milliseconds.

In Figure 7b, we see that the update process in the Nomad system takes significantly longer than in the Kubernetes system. According to our setting of the *maxParallel* parameter, between 8 and 10 containers are deployed at any time. Similar to the Kubernetes system, most request failures occur toward the end of the update process. However, the total number of failed requests is significantly lower. Our investigations motivate to set minimum healthy times/cooldown times for rolling updates to ensure that sufficient containers are always accessible and routing tables can be updated.

We look at the empirical load distribution over individual instances to understand how load balancing works in the benchmarked systems. Table 4 shows the average number of requests received by an instance per second and its standard deviation. The numbers have been measured in the rolling update experiments. However, they have also been confirmed in the context of the failure recovery experiments from Section 5.3. The average request rate is calculated as follows: Every test container counts the number of user requests it receives. In addition, it captures both its start and termination time. The difference between termination and start time is the total instance runtime. To calculate the request rate, we divide the total received requests by the instance runtime.

We see significant differences between Kubernetes and Nomad in terms of load balancing. Kubernetes distributes requests much more inequally among instances than Nomad. This indicates a different load-balancing algorithm. Nomad is much closer to an equal distribution with a significantly lower standard deviation. There is no significant difference between different cluster environments. This shows that COFFEE can capture empirical results for load balancing metrics. This enables investigations of load balancers that are not open-source, for instance, the Google Cloud Load Balancer. Furthermore, one can compare different load-balancing options, for example, as offered by Nomad.[10] As mentioned in Section 5.1, we use HAProxy load balancing for our Nomad test systems, but other open-source load balancers might be future benchmarking targets.

---

[9]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment

[10]https://developer.hashicorp.com/nomad/tutorials/load-balancing/load-balancing

6 DISCUSSION

In this section, we give a short summary of our main findings from Section 5 and point out limitations and open challenges. All in all, we conclude from our case studies that COFFEE is able to capture several performance metrics for container orchestration frameworks. We could detect performance differences for the same orchestration framework running on different hardware (self-hosted vs. Google Cloud environment), as well as for different orchestration frameworks running on the same hardware (Kubernetes vs. Nomad). Our results indicate that Kubernetes outperforms Nomad in many scenarios, but more experiments in different environments are needed to solidify this statement.

This leads to the limitations and open challenges of this work. Our approach and the COFFEE framework provide a basis for future work and numerous case studies in the area of benchmarking of CO frameworks. However, our work does not yet represent a complete benchmark as defined by the latest empirical standards [11]. The main missing points are representative workloads for container orchestration frameworks. In the context of COFFEE, this means we need realistic test scripts. Our case studies cover the basic functions of CO frameworks. More enhanced studies would be enabled if more data from production workloads of CO frameworks were available.

One aspect we have not yet evaluated in this paper is the size of the test cluster and the scalability of our approach. The influence of the number of nodes on metrics like the container start time or load distribution is still to be investigated. We reduced the number and size of requests to the COFFEE controller as far as possible in the design phase. However, there is still proof needed that COFFEE can handle complex test scripts in large test clusters. Further optimization or redundancy of the COFFEE controller might be necessary. Another current limitation is that only one type of test container can be deployed in the cluster. In future work, one could introduce different types of testing containers, for instance, with different resource requirements, to evaluate the impact on readiness times. As discussed in Section 3.1, we limited our case studies to the comparison of different test environments and orchestration frameworks. We left the use case of tuning an orchestration platform open for future work. Changing the configuration of the CO frameworks might change the benchmarking results presented in this study, which is why our results from the case studies cannot be generalized. However, we decided to show the usage of COFFEE with different orchestration platforms and test environments to underline COFFEE's broad use cases. For studies comparing different orchestrator configurations, we refer to related work in Section 7.

In our case studies, we have not yet covered all the core tasks of CO frameworks, especially autoscaling and placement. COFFEE can track placement decisions and also supports autoscaling experiments by supporting dynamic loads or reconfiguration of the load endpoint. These two tasks, in particular, have the characteristic that they are commonly evaluated with application-specific metrics like response times and do not offer directly interpretable and application-agnostic metrics. For these purposes, specialized frameworks such as Theodolite [12] for scaling might be preferred in some use cases. The full potential of COFFEE could be achieved by including system-level metrics and orchestrator-specific data, such

(a) Google Kubernetes Engine
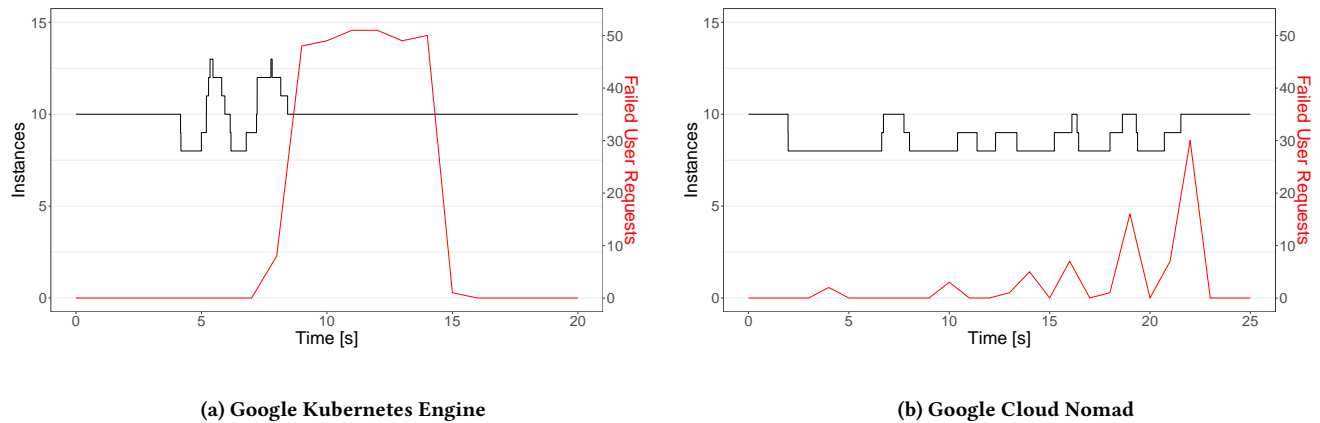
(b) Google Cloud Nomad

Figure 7: Detailed view on rolling updates for exemplary runs.

as log data, to track also fine-grained metrics, such as the runtime of load balancing algorithms [39]. However, including more metrics raises the question of how to aggregate these metrics, e.g., to a total score for the whole CO framework. More measurements with more frameworks and in different environments must be conducted to propose such an overall score. In this paper, we show the usage of COFFEE with its minimum requirements to make clear that our approach does not rely on advanced monitoring capabilities and the presence of system-level or orchestrator-specific metrics.

We reduce statistical errors by repeating all measurements 100 times. However, as known from previous studies [22, 27], measurements in public clouds are always subject to variability, and results in other settings, for instance, other compute regions, might be different. More empirical results must be collected to allow further claims on the performance of different cloud environments. Note that we selected nodes with more computing resources than required to avoid hardware bottlenecks. In test runs of our workloads, CPU and memory utilization of all nodes stayed far below 50%. In our case, the most significant source of uncertainty is the time synchronization between the nodes. Some operations rely on timestamps from different machines. This influence is especially significant for network measurements. We use network time protocol-based (NTP) synchronization between hosts in all test clusters. Better synchronization methods like precision time protocol (PTP) are not supported on Google Cloud virtual machines as they do not fulfill specific network interface requirements.

7 RELATED WORK

With the emerging trend of containers and the release of Docker and Kubernetes, there have been several studies that have made comparisons between individual orchestration frameworks. Al Jawarneh et al. [17] focus on provisioning and availability metrics for Kubernetes, Docker Swarm, Mesos, and Cattle. Truyen et al. [35] conduct a comprehensive feature comparison study between Docker Swarm, Mesos, Kubernetes, and other orchestration frameworks. Pan et al. [26] compare the performance properties of Kubernetes and determine its influence on the execution time of cloud applications. Fayos-Jordan et al. [8] compare Kubernetes and Docker Swarm for IoT use cases. All these studies have in common that they were

conducted before the extensive adoption of CO frameworks in production systems. Consequently, they do not cover the latest improvements, and some results are even outdated, e.g., the classic Docker Swarm features are now deprecated.[11]

Besides the papers investigating container orchestration frameworks, some works analyze selected orchestration tasks and their performance. Bozoki et al. [4] focus on the resilience testing of Kubernetes and especially examine pod restart times. Similar to our work, fault injection is used to force restarts. Kjorveziroski and Filiposka [19] compare different deployment options of Kubernetes for serverless applications based on start and response time metrics. Multiple other works [3, 32] investigate performance and resource consumption metrics for different Kubernetes-like frameworks. In the pre-container era, performance analysis approaches for cloud environments targeted provisioning times and throughputs [7, 16, 30]. Yu et al. [39] evaluate load balancing algorithms in the IoT sector for interconnected microservices. They explicitly consider the time to decision, the maximum runtime of the load balancing algorithms, as a performance metric. Amaral et al. [1] evaluate the performance of container-based microservices and target the issue of network performance, throughput, and latency. In the model-based approach of Medel et al. [23], the used bandwidth of containers within one or more Kubernetes pods is analyzed. The works of Zeng et al. [40] and Bankston and Guo [2] focus on the empirical evaluation of different container networking solutions.

Lei et al. [21] focus on the scalability of Kubernetes clusters. Pod startup times, latencies of API callbacks, and other metrics are analyzed in differently-sized clusters. Henning and Hasselbring [12] focus on scalability benchmarking of cloud-native applications. Therefore, they also provide tooling for Kubernetes and conduct experiments on the Google Cloud Platform. Other benchmarking tools, especially for Kubernetes, include kubestr [5], which evaluates the performance of different cluster storage options, and kube-bench [29], which is concerned with the security evaluation of Kubernetes clusters. K-Bench [38] is a framework for assessing the performance of Kubernetes' control and data plane with support for pod start, networking, and I/O metrics. Nomad has not been

---

[11]https://docs.docker.com/engine/deprecated/

investigated much in the scientific literature, only in conjunction with the deployment of network services [36, 37]. To the best of our knowledge, no recent work focuses on as many CO tasks as this work and supports different container orchestration frameworks.

8 CONCLUSION

CO frameworks play essential roles in modern cloud computing paradigms such as cloud-native or serverless computing and perform many performance-relevant tasks in parallel. To enable comprehensive benchmarking of CO frameworks, it is therefore important to cover the large variety of orchestration tasks. Furthermore, a generic approach must be chosen to allow comparing different frameworks and technology stacks. This paper presents the first benchmarking approach for CO frameworks that covers several CO tasks and has been tested with multiple CO frameworks. Therefore, we first set a benchmarking scope by defining tasks and requirements for CO frameworks. Subsequently, we describe performance metrics and introduce COFFEE, a benchmarking framework that allows the definition and execution of complex benchmarking scenarios. We demonstrate the relevance of the problem and the usefulness of our approach in case studies with Kubernetes and Nomad in a self-hosted environment and on Google Cloud VM instances. Our studies found significant performance differences, e.g., regarding container readiness times and rolling updates.

As future work, more empirical studies must be conducted to evaluate and extend our approach. With sufficient data and experience, realistic scenarios and aggregated metrics, such as a total performance score of CO frameworks, can be defined. Potential experiments of interest include systematic studies on the performance of CO frameworks in public and private clouds, comparison, ranking, and tuning of different frameworks and configurations, and regression testing of CO frameworks. Our work thus provides the basis for various enhanced studies, and with COFFEE, a ready-to-use and extensible benchmarking framework for CO frameworks is available.

REFERENCES

[1] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. 2015. Performance Evaluation of Microservices Architectures Using Containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*. 27–34.
[2] Ryan Bankston and Jinhua Guo. 2018. Performance of Container Network Technologies in Cloud Environments. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*. 0277–0283.
[3] Sebastian Böhm and Guido Wirtz. 2021. Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes.. In *ZEUS*. 65–73.
[4] Szilárd Bozóki, Jenő Szalontai, Dániel Pethő, Imre Kocsis, András Pataricza, Péter Suskovics, and Benedek Kovács. 2020. Application of Extreme Value Analysis for Characterizing the Execution Time of Resilience Supporting Mechanisms in Kubernetes. In *Dependable Computing - EDCC 2020 Workshops*. Springer International Publishing, Cham, 185–199.
[5] Git Repository by Kastenhq. 2022. *Kubestr*. Retrieved October 14, 2022 from https://github.com/kastenhq/kubestr
[6] Emiliano Casalicchio. 2019. *Container Orchestration: A Survey*. Springer International Publishing, Cham, 221–235.
[7] Mohan Baruwal Chhetri, Sergei Chichin, Quoc Bao Vo, and Ryszard Kowalczyk. 2013. Smart CloudBench – Automated Performance Benchmarking of the Cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*. 414–421.
[8] Rafael Fayos-Jordan, Santiago Felici-Castell, Jaume Segura-Garcia, Jesus Lopez-Ballester, and Maximo Cobos. 2020. Performance comparison of container orchestration platforms with low cost devices in the fog, assisting Internet of Things applications. *Journal of Network and Computer Applications* 169 (2020), 102788.
[9] Cloud Native Computing Foundation. 2022. *CNCF Annual Survey 2021*. Retrieved October 13, 2022 from https://www.cncf.io/reports/cncf-annual-survey-2021/

[10] VMware Glossary. 2022. *What is container orchestration?* Retrieved October 14, 2022 from https://www.vmware.com/topics/glossary/content/container-orchestration.html
[11] Wilhelm Hasselbring. 2021. Benchmarking as Empirical Standard in Software Engineering Research. In *Evaluation and Assessment in Software Engineering* (Trondheim, Norway) *(EASE 2021)*. Association for Computing Machinery, New York, NY, USA, 365–372.
[12] Sören Henning and Wilhelm Hasselbring. 2022. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering* 27, 6 (2022), 1–42.
[13] Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. 2015. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 46–56.
[14] IBM Cloud Learn Hub. 2021. *Container orchestration*. Retrieved October 14, 2022 from https://www.ibm.com/cloud/learn/container-orchestration
[15] Red Hat Inc. 2022. *What is container orchestration?* Retrieved October 14, 2022 from https://www.redhat.com/en/topics/containers/what-is-container-orchestration
[16] Alexandru Iosup, Simon Ostermann, M. Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. 2011. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems* 22, 6 (2011), 931–945.
[17] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. 2019. Container Orchestration Engines: A Thorough Functional and Performance Comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 1–6.
[18] Asif Khan. 2017. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Computing* 4, 5 (2017), 42–48.
[19] Vojdan Kjorveziroski and Sonja Filiposka. 2022. Kubernetes distributions for the edge: serverless performance evaluation. *The Journal of Supercomputing* (2022), 1–28.
[20] Samuel Kounev, Klaus-Dieter Lange, and Joakim Von Kistowski. 2021. *Systems Benchmarking: For Scientists and Engineers*. Springer.
[21] Qing Lei, Weidong Liao, Yingtao Jiang, Mei Yang, and Haifeng Li. 2019. Performance and Scalability Testing Strategy Based on Kubemark. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. 511–516.
[22] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (apr 2016).
[23] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. 2016. Modelling Performance & Resource Management in Kubernetes. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*. 257–262.
[24] Mohamed Mekki, Nassima Toumi, and Adlen Ksentini. 2022. Microservices Configurations and the Impact on the Performance in Cloud Native Environments. In *2022 IEEE 47th Conference on Local Computer Networks (LCN)*. 239–244.
[25] HashiCorp Nomad. 2022. *Who Uses Nomad*. Retrieved October 14, 2022 from https://www.nomadproject.io/docs/who-uses-nomad
[26] Yao Pan, Ian Chen, Francisco Brasileiro, Glenn Jayaputera, and Richard Sinnott. 2019. A Performance Comparison of Cloud-Based Container Orchestration Tools. In *2019 IEEE International Conference on Big Knowledge (ICBK)*. 191–198.
[27] Arnaldo Pereira Ferreira and Richard Sinnott. 2019. A Performance Evaluation of Containers Running on Managed Kubernetes Services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 199–208.
[28] Maria A Rodriguez and Rajkumar Buyya. 2019. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience* 49, 5 (2019), 698–719.
[29] Aqua Security. 2022. *Kube-bench*. Retrieved October 14, 2022 from https://github.com/aquasecurity/kube-bench
[30] Marcio Silva, Michael R. Hines, Diego Gallo, Qi Liu, Kyung Dong Ryu, and Dilma da Silva. 2013. CloudBench: Experiment Automation for Cloud Environments. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*. 302–311.
[31] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. 2022. Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 105–115.
[32] Sergii Telenyk, Oleksii Sopov, Eduard Zharikov, and Grzegorz Nowakowski. 2021. A Comparison of Kubernetes and Kubernetes-Compatible Platforms. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Vol. 1. 313–317.
[33] Eddy Truyen, Matt Bruzek, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. 2018. Evaluation of Container Orchestration Systems for Deploying and Managing NoSQL Database Clusters. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 468–475.
[34] Eddy Truyen, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. 2019. Performance Overhead of Container Orchestration Frameworks for Management of Multi-Tenant Database Deployments. In *Proceedings of the 34th ACM/SIGAPP*

Symposium on Applied Computing (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 156–159.

[35] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. 2019. A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks. *Applied Sciences* 9, 5 (2019).

[36] Alexandros Valantasis, Nikos Makris, and Thanasis Korakis. 2022. Orchestration Software for Resource Constrained Datacenters: an Experimental Evaluation. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 121–126.

[37] Alexandros Valantasis, Nikos Makris, Christos Zarafetas, and Thanasis Korakis. 2021. Experimental Evaluation of Orchestration Software for Virtual Network

[38] VMware. 2022. *K-bench*. Retrieved January 23, 2023 from https://github.com/vmware-tanzu/k-bench

[39] Ruozhou Yu, Vishnu Teja Kilari, Guoliang Xue, and Dejun Yang. 2019. Load Balancing for Interdependent IoT Microservices. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 298–306.

[40] Hao Zeng, Baosheng Wang, Wenping Deng, and Weiqi Zhang. 2017. Measurement and Evaluation for Docker Container Networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 105–108.

Functions. In *2021 IEEE Wireless Communications and Networking Conference (WCNC)*. 1–6.